4. DII COE/SHADE Database Concepts

SHADE uses database segmentation and Shared Data Servers (SDS) as the primary underlying mechanisms to enable data sharing. Packaging data into database segments and installing them on an SDS allows multiple organizations and functions to share single copies of a DBMS (i.e., one per physical data server) in the same way that packaging applications into segments allows software systems for multiple functions to coexist on the same workstation. At the same time, identifying database segments as Shared or Universal allows explicit sharing of data among software systems, applications, and their user communities. Database segments provide a convenient way for organizations to load SDSs with the data structures and values that users require. This packaging technique will also make it easier for data administrators to collect required metadata on implemented structures and to physically reorganize (multi-segment) databases in ways conducive to distribution and replication.

The function of a COE SHADE Database Server (SDS) together with the databases it manages is to provide information to users through applications that access the databases, and to support system and database administrators' maintenance functions. The operations of an SDS involve the database server, the databases/database segments managed by it, and the applications that access one or more databases. The discussion that follows addresses the operational roles of each.

An SDS provides data management services to its client applications. In order to be useable, it must constitute a stable, reliable operating environment that developers can design for. Database services include tools to support the management, by a site administrator, of users' discretionary access to databases based on the applications they are permitted to use. This is governed by the following principles.

- Users will not need access to all applications.
- Applications will have multiple levels of database access that can be granted to users.
- When access to an application is granted to or revoked from a user, the corresponding database permissions are also granted or revoked.

SHADE database services within the COE are implemented as a federation of application-owned (Unique) and common (Shareable, Shared, and Universal) database segments. Application segment developers control the data and structures that are specific to their Unique segments and can change the data or their structure when necessary. The configuration of Shared and Universal database segments is controlled by the appropriate joint configuration management authority. All these databases reside on the SDS that provides services to the applications, acting as database clients, within the network. The databases within a particular SDS are isolated from each other, physically and logically, by being placed in separate storage areas and by being owned by different DBMS accounts. Database developers sustain this isolation by defining one or more database accounts to own their data objects and by allocating those objects to the owner accounts they have created.

This configuration, using a disk controller and drive analogy, is shown in Figure 4-1. The core database configuration, containing the DBMS Data Dictionary and associated system information,

is part of the COE and is represented by the System Database. All other databases, whether provided by DISA, a developer, or some other agency, are included as 'component' databases under the management of the SDS. The set of component databases available from a particular SDS is determined by the set of applications that server's database is expected to support.

From the DBMS perspective, all databases are shared assets, whether they are common or not, because they are accessed by multiple concurrent users. They are also dynamic because their data changes even while structure remains static. Databases may be interdependent. Databases depend on the COTS DBMS service and are built within its constraints. Databases can be accessed by applications other than those written by the database developer. While database applications today are usually written by the database's developers, this will be less true in the future as SHADE data object reuse increases.

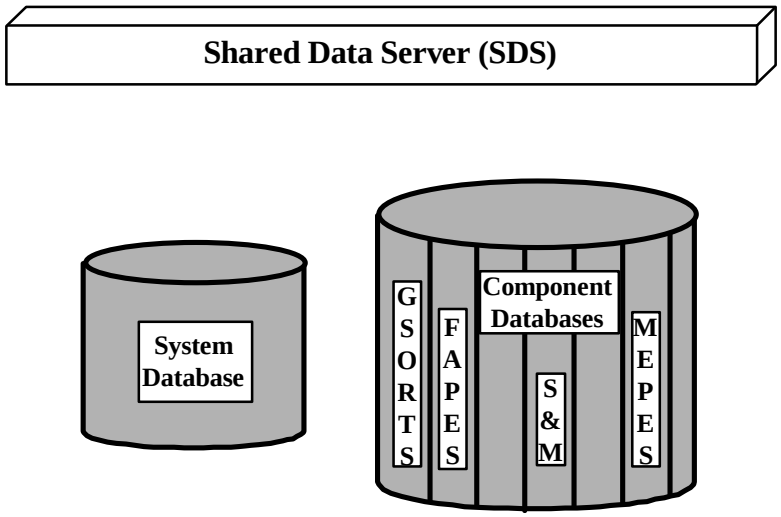

Figure 4-1: Shared Data Server Architecture

Applications that use databases to manage their information are the interface between users and the data. Some applications use their databases interactively, in a transaction-processing mode, to perform the work for which they have been designed. Others have a single process that writes data for many readers. Some pull data from remote sources directly to replace existing data. They then allow read-only access to that remotely provided information.

Users connect to the SDS through the client applications, possibly in multiple sessions. Each session must behave as if it is isolated from the rest of the system and knows of no data other than that belonging to the application it is executing.

4.1 Constraints on Database Developers

The developers of databases and applications accessing databases must conform to the COE database server environment so they do not bypass its features. Conformance also limits the likelihood of data corruption. The combination of the SDS configuration and the developers' implementations must ensure two things. First, each connection of a user to a database through an application must function in the proper context for that application and database. Second, each user's connection to a database must not interfere with any other user's connection to the same or any other database.

The development and integration standards for COE databases support an evolving configuration of database services. Using GCCS as an example, in version 0 each GCCS application had its own database and database management system. Commencing with GCCS 1.1, the separate database servers were replaced by a single-server running a single-instance of the database management system. Each application retained ownership of its database within that instance, but shared the DBMS service with the other applications' databases. The next step was to have a database segment on the server that is accessible from multiple application databases. For example, suppose two application databases need a country-code table. In the prior step, each database would have its own version of the country-code table, which might be identical. In this step, a single copy of the country-code database segment would be on the SDS and accessed by both applications. Thus, the SDS provides shared, concurrent access to multiple databases and database segments with varying degrees of autonomy. COE-based systems are to follow the same approach as that pioneered by GCCS.

The principal reason for this change in GCCS was to meet DOD's information availability requirements. The multiple instance configuration split information among data applications that were uniquely configured to support the needs of specific mission applications. The single-server, single-instance data management service provided by GCCS conserves system resources by not requiring multiple copies of the DBMS to be executing and eases system management by providing a single point-of-entry for database management services. That single point of entry also simplifies application development. However, that is not the only method for implementing SHADE data services. COE systems may implement a configuration that distributes the database over a LAN/WAN for survivability or to distribute the processing load. Multiple DBMS instances may be used for data isolation or to separate different user communities on the same server. Regardless of the specific database server configuration, SHADE requires that information be treated as a DOD corporate resource, not something owned by applications. The benefits that come with the central service does limit the freedom of developers by requiring that they implement databases compatible with the larger multi-database environment. In addition, the increased complexity of a multi-database system could overburden the operational sites' system and database administrators unless it is implemented consistently. This again limits developers by constraining their databases to function within a consistent administrative framework.

The principal consideration for developers is that their applications and databases no longer have exclusive use of the database management system. Instead of being an application-specific data management tool, the DBMS is a central service that supports all applications' databases. As a

result, developers cannot customize or tune the DBMS to the particular behavior of any single application. Any such modifications to the DBMS will inevitably affect other applications and databases. Similarly, the individual component databases are no longer the sole occupants of the DBMS. Developers must implement their applications, constraints, and component databases so that they do not interfere with others sharing the same DBMS. Further, because there are multiple databases in the DBMS, applications can connect improperly to other databases. Developers must ensure that their applications connect only to the database they intended to use. They must also design their databases to maintain their own integrity without reference to external applications.

In order for component databases to plug into and play properly on an SDS, they must conform to the standards defined herein. The objective is to support the independent development of maintainable databases that will function reliably within the larger multi-database system. This release of the *I&RTS* has extended the COE tool set to include tools that deal specifically with integration problems related to multi-database environments.

Developers must implement their databases such that the operational sites' administrators can manage the collection of databases. If system and database administrators are required to manage multiple databases, each with its own integrity rules and access methods, their jobs quickly become impossible. This means developers must adhere to common implementation methods.

4.2 Database Integration Requirements

The SHADE Database Server is the COE component that provides shared data management within COE-based systems. Regardless of the COTS DBMS used to provide database services, its functions within the system remain the same:

- Support independent, evolutionary implementation of databases and applications accessing databases
- Manage concurrent access to multiple, independent, and autonomous databases
- Maintain integrity of data stored in the DBMS Server
- Provide discretionary access to multiple databases
- Sustain client/server connections independent of the client application's and database server's hosts
- Support distribution of databases across multiple hosts with replicated data and with distributed updates
- Provide maintainability of users' access rights and permissions
- Support backup and recovery of data in the databases.

In addition, database services within the COE are not restricted to a single vendor's DBMS. As a result, developers must implement their databases such that dependence on any particular DBMS vendor's product is limited. The discussion that follows provides more detail on each of these general requirements.

4.2.1 Evolutionary Implementation

The goal of evolutionary implementation is to be able to incrementally develop, field, and improve software and information services. This "build a little, test a little" philosophy applies to databases as well as applications. In the database context, the objective is to field the latest and best information structures and contents, and to progressively reduce the number of structural variants representing the same entities and relationships. Databases and applications should be able to evolve independently in principle, but in practice this is tempered by the dependence of applications on the database's structure. In addition, component databases are dependent on some unique DBMS features for their implementation.

Database developers can still support evolutionary implementation by maintaining the modularity of their component databases. To achieve this goal, component databases must first coexist within the server without corrupting each other's data. This does not simply require isolating databases from each other; it requires that all actions across database boundaries be intentional and

documented. The COE/SHADE architecture requires that segments not modify other segments. The same applies to component databases modifying or extending other database segments. When a database segment does have a dependency on some other component database, that dependency will be kept in a separate segment.

Component databases are dependent on the DBMS used for the SDS. The specific commands used for their implementation within the DBMS and the environment it provides are both defined by the DBMS vendor. Database developers must be careful in their use of vendor-specific features so they do not create unintended dependencies on specific database management systems or, more importantly, particular versions of the DBMS, while still taking advantage of the database server's capabilities. To accomplish this, developers shall separate DBMS-specific code from that which is transportable. See Appendix F of this document for information on vendor products. Additional information and guidance on SHADE-specific issues can be found in the SHADE Architecture and related documents.

The same constraints on databases also apply to applications accessing those databases. Application developers must ensure that applications connect through regular, documented APIs and shall not assume the use of particular DBMS versions. This does not prohibit developers from designing to the current version of a COE-compliant DBMS, using vendor-supplied tools that are part of the COE, or from accessing objects in other database segments. It prohibits developers from embedding DBMS vendor's runtime libraries or environment variables in the application segment. For example, developers should not provide their own `coraenv` script in the application segment because it creates an implicit version dependency on that version of the Oracle RDBMS. In addition, this example interferes with the DBA's management functions.

The key to managing the evolution of component databases and the applications that use them is documenting their interrelationships. Applications' dependencies on databases shall be documented so that database-segment version changes can be tested with the applications. The component database's dependency on the DBMS will also be documented for the same reason. If developers use DBMS vendor-supplied tools to implement applications, the dependency on the tools will be documented. When applications or component databases access data objects belonging to other component databases, the dependency among the databases shall be documented as well. These dependencies are documented under the `Database` and `Requires` descriptors of the segment's `SegInfo` file. See Chapter 5 for more information.

When one developer is responsible for both applications and databases, the management of such interdependencies is simplified. Database segments and associated application segments will usually be delivered at the same time and installed together. When separate developers are responsible for databases and applications, however, careful coordination between the two developers will be required. As the database federation evolves, it is likely that component database segments will be upgraded before applications that access them. When applications are affected by component database segment modifications, legacy views may be provided as directed by the cognizant authority for the segment. Such views will be read-only, but can allow query tools to continue to function until they are modified to work with the re-engineered database.

4.2.2 Database Segmentation

Another issue with respect to modularity is that of subdividing a database into coherent segments. If a database is only supporting one application, then it might make sense to implement a single Unique database segment. However, if the database supports more than one application then the developer should determine if the database should be developed with one or more Shared database segments and Unique segments. The advantage of multiple shared database segments is that the segments are more granular and an SDS can be configured to support the data requirements of mission applications without having to carry superfluous data services. A disadvantage of multiple shared database segments is the management of database object dependencies such as foreign key constraints. These inter-segment dependencies complicate the installation and deinstallation of database segments.

In the course of determining which data objects will be grouped into a database segment, developers need to consider several factors:

- Data Objects that can be conveniently managed as a unit,
- Data Objects that are needed together to support a functional area,
- Common sources or providers of data,
- Data object interdependencies, and
- Frequency of update.

Modularity can be enhanced by allocating data objects among Shared and Unique database segments. A Shared database segment contains data objects that are intended for use by multiple applications or other data stores. A Unique segment's objects are specific to the applications contained in a specific software segment. Additionally, the developer should investigate the SHADE repository for existing Universal and Shared database segments for potential reuse. For example, if the SHADE repository contains a Universal database segment for country codes, then it may be possible to remove the table(s) defining country codes from the proposed segment and use the existing country code database segment. In the case of a legacy system, a view may need to be created to the Shared or Universal database segment until the application can be modified. Divide the remaining tables in the database along functional boundaries to form segment groupings (i.e., neither unique nor replaceable by an existing database segment). Potentially Shared database segments should be registered in the SHADE repository. The outcome of this process should be a set of one or more database segments with their corresponding groups of identified database objects. Specific guidelines for creating database objects are found in subsection 4.3.

The database objects in a Shared database segment are common to many applications residing in different segments. Shared database segments prevent duplication of widely used or required database objects, such as reference tables, and procedures, such as validation or conversion routines. They also support interoperability at the data level by standardizing key cross-reference fields. The objects in a Shared database segment must be accessible to many applications, regardless of which database they reside in, and may support other database segments that are then dependent on that Shared database segment. Such segments will often provide generic read-

only or read/write database roles (see subsection 4.3.5) to support their use by other segments. In this context, the only distinction between a Shared and a Universal segment is the organizational level at which their contents are managed.

The database objects in a Unique database segment are not open to, nor intended to support, multiple software or data store segments, but are used only by a particular software segment. This software segment owns, controls, and depends on its own database segment, and no other software segment does. Thus, a Unique database segment usually contains the database tables, triggers, and procedures that support specific, intrinsic functions of a software segment; it has no data of value to any other segments.

Dividing data in this manner simplifies the system integration effort. When a change is made to a Shared database segment, all developers of applications that access that segment must be notified and must be given time to adjust their application segments. Otherwise the Shared database segment must incorporate legacy views to support the applications until they can be modified. Changes to Unique database segments, however, require no coordination as only the applications in the dependent segment are affected. In addition, legacy views are seldom required as the applications and their database segment, both usually maintained by the same developer, will be modified at the same time.

Developers should also consider the frequency of updates against data tables when defining their database segments. Separating static reference tables from those that are dynamic allows more flexible system and database administration. The separation may be accomplished by placing the static objects in their own database segment, or by creating static objects in a separate storage area (e.g. an Oracle tablespace for read-only tables) within the segment. The appropriate method will depend on the target DBMS. See subsection 4.3.2 for storage allocation methods and Appendix F for implementation information specific to each vendor's product.

4.2.3 Managing Multiple Databases

The COE database architecture is a federation of databases with varying degrees of autonomy. Federated means that the component databases are collocated and share DBMS resources. They process data cooperatively but are not part of an overall schema. They may use Shared and Universal database segments. In some cases they may also share or exchange data. Autonomous means that each database remains an independent entity. Individual databases may be modified or upgraded without reference to others (e.g., segments may be added or deleted to support the functionality of the applications that use the database). Individual database segments within a database may not be changed without reference to others unless they are unique segments. Developers are also responsible for maintaining their own data access and update rules.

The federated architecture provides the same modularity within the SDS that mission-application segmentation does for the user interface. The set of databases available from any particular SDS is tailored to the information needs of the individuals using that server. Database segments that are not needed can be omitted. This may appear to conflict with SHADE's stated goals of improving interoperability through data standardization. In fact, it supports those goals by separating the

data that can be shared from those that should not be. As a result, developers and data stewards can concentrate their efforts on standardizing those data where there is the most benefit in terms of quality and interoperability.

In order for this to work, each component database must be implemented in a self-contained manner. This is not to say that a database supporting a set of applications should be self-sufficient. One goal of SHADE's modular database implementation is to limit the redundancy of information among component databases. Developers should not incorporate information in component databases that is already available from other, existing database segments. Instead, being self-contained means that each component database must contain all information needed to manage its objects and maintain their integrity. The issues involved in implementing this are discussed in the next subsection.

4.2.4 Data Integrity

Data integrity addresses the protection of the information stored within a database management system. There are three general circumstances that must be addressed.

1. The prevention of accidental entry of invalid data.
2. The security of the database from malicious use.
3. The protection of the database from hardware and software failures that may corrupt data.

Implementation of appropriate data integrity measures is the responsibility of the database developers using the features of the DBMS.

The SDS is responsible for preserving the integrity of each component database and for preventing connections between an application and data that belong to any other application. COE-based systems may well be secure systems that contain and process classified data. The database management component must conform to the security policies and practices of the overall program. Otherwise, the SDS supports the data access restrictions and integrity assumptions incorporated in each database.

The SDS provides the basic functionality expected of a DBMS. It ensures the recoverability of failed transactions or of a crashed system. The atomicity, concurrency, isolation, and durability of database transactions are the responsibility of the applications accessing the server. However, supporting these transaction properties is the server's responsibility. Developers must pay special attention to transaction isolation because of the multi-database configuration of most COE-based systems.

Database developers are responsible for defining and implementing the integrity constraints of their databases. The SDS is responsible for enforcing the developers' integrity constraints when they are defined within the database. Application developers must ensure that their applications connect properly to their databases and do not connect improperly to anyone else's database segments. Adoption of these practices protects all applications' data and allows the SDS to maintain all databases reliably.

Within a component database the implementation of data integrity takes the form of what are often called constraints and business rules. In the current context, constraints are defined as the rules within the database that govern what values may exist in an object. Business rules are those rules within the database that govern how data is updated and what actions are permitted to users.

Until recently, commercial database management systems were limited in their ability to support the variety of constraints and business rules that may be needed in a database. As a result, most constraints and business rules of legacy DII databases have been implemented in the applications, not the database. Because of the federated database architecture and because the applications that maintain those databases are also developed independently, it is difficult to ensure uniform and consistent enforcement of those rules and constraints by a DII COE SDS.

To avoid problems with constraint enforcement in the DII environment, developers must place their business rules and constraints in their databases rather than their applications. This keeps control of data maintenance access in the hands of the developers where it belongs and ensures that constraints cannot be bypassed. Developers have the knowledge of their constraints and business rules; DBAs and users do not.

The reason for placing constraints in the database is shown in Figure 4-2. Application One and its associated component database were implemented with business rules and constraints in the application. Application Two placed those constraints in the database. When a third application (Browser) accesses both databases, it is unaware of Database One's business rules because they are inaccessible. If this application, which could be a user-developed query tool, modifies Database One, it could corrupt the database out of ignorance.

Placing the business rules and constraints in the database promotes client/server independence. The efficient implementation of constraints and business rules will have to make use of the DBMS capabilities. If these rules are in the component database, the application is less dependent on the COTS DBMS product. Also, this approach can reduce network communications loading by allowing the DBMS, rather than the application, to enforce the rules within the database. Checking rules within the database avoids passing multiple queries and their results over the network between the DBMS and the application.

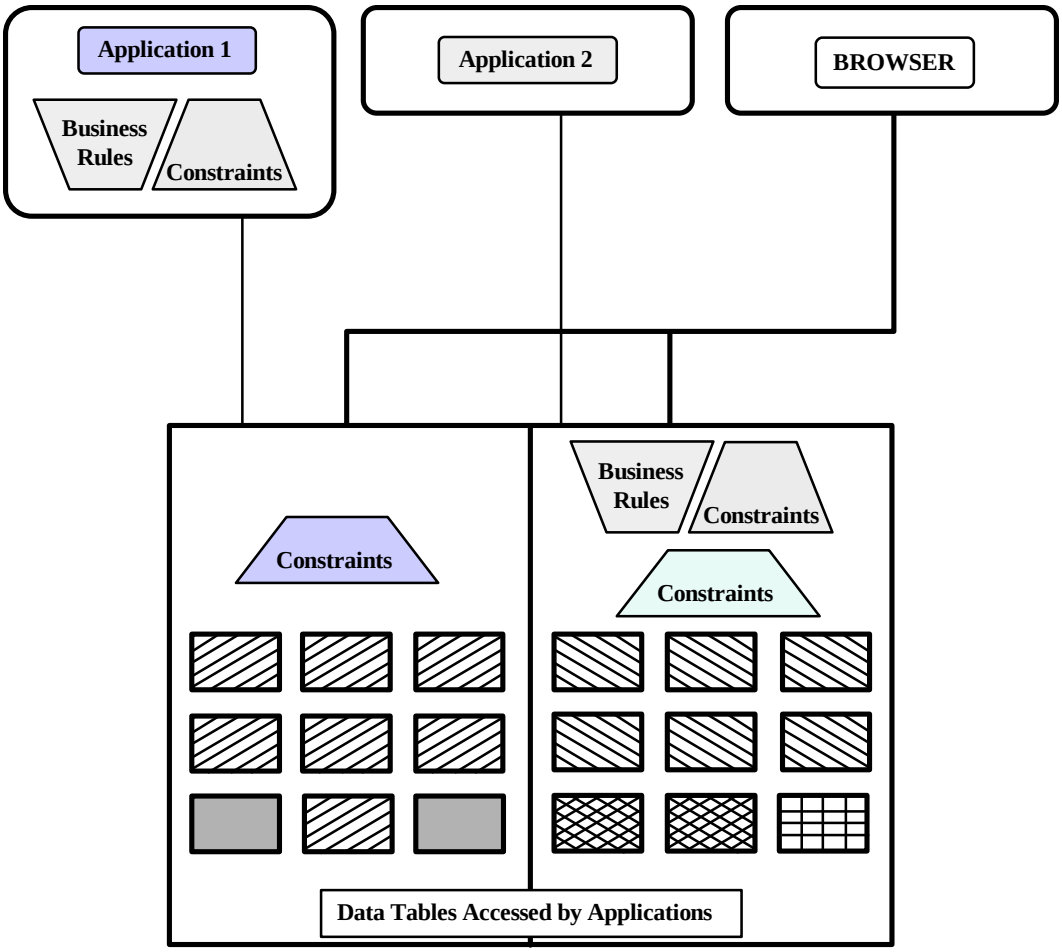

Figure 4-2: Business Rules and Constraints

4.2.5 Discretionary Access

Discretionary access addresses the selective connection of users to databases through applications. Database access is discretionary because not all users have the same permissions to use applications. The objective is to ensure that users' database connections operate in the proper context for the applications. Users must be able to operate several different applications at the same time. The DBMS server must effect each application's accesses to different sets of data objects. This means permission to access to specific tables and the mode (read or write) of that access. Because several databases exist on the SDS, each application must be written to access only the database it belongs with; it must be unable to access tables belonging to some other application for which it does not have access privileges. Each user-application connection will have only the permissions needed for that context.

In this context, DII databases can be broadly characterized as either public or private. A *public* database is intended to be generally available and, in most cases, access to it will be given to all users of a particular system. Public databases are usually read-only. Access to a *private* database is discretionary, not general, and is usually restricted to a small group of a system's users. That

system's administrators must specifically grant individuals access to a private database. Private databases often have users with read/write permissions and users with read-only permissions. A public database will be composed of Shared database segments as defined in subsection 4.2.1 A private database may also contain Shared database segments, but the data itself needs to be more closely controlled than the public database. The public/private and Shared/Unique categories address different issues. The distinction has to do with user access in the former case and with configuration control in the latter case. A Shared database has many developers writing applications against it; its schema cannot be easily modified. A public, application-owned database would have one developer but many users; its schema could be changed without affecting other developers.

There are three components to the discretionary access issue: Session Management, Discretionary Access Control, and Access Management. The first refers to the DBMS' ability to keep different connections separate. The second addresses the context of an individual connection. The third deals with the requirements of system and database administrators to manage the accesses that are provided to users. Without the correct functioning of all three components, data integrity and consistency can be compromised.

In order for a COE system to be useable, it must provide support to systems administrators as they manage users' discretionary access to subsets of applications and databases. This means that the approach taken in supporting access management must fit with overall system administration and security policy.

4.2.5.1 Session Management

A *database session* is an individual connection between an application and the database management system. It is the means by which the SDS isolates one user's activities working with an application from all other users that are connected to the DBMS. In this context autonomous applications such as message processors are also database users.

The SDS is responsible for session management as shown in Figure 4-3. In this example, two users are connected to the SDS. The first has two sessions with application A and one with application B. The second has a session with application B and one with application C. The SDS maintains five separate sessions. Two sessions are connected to component database A, two to component database B, and one to component database C; no session is connected to component database D.

Note that each different execution of an application is considered a separate session and is functionally isolated from other executions of the same application. Thus, when User 1 starts two separate instances of application A, the DBMS treats them as different sessions (A1 and A2). This ensures that changes being made in different sessions propagate correctly and do not corrupt data accessed by other sessions.

The key points with respect to session management are that the DBMS, in managing connections, provides sessions to isolate each one from all others. Isolation facilitates transaction management

and system recovery. It also supports the traceability of database transactions to the user and application.

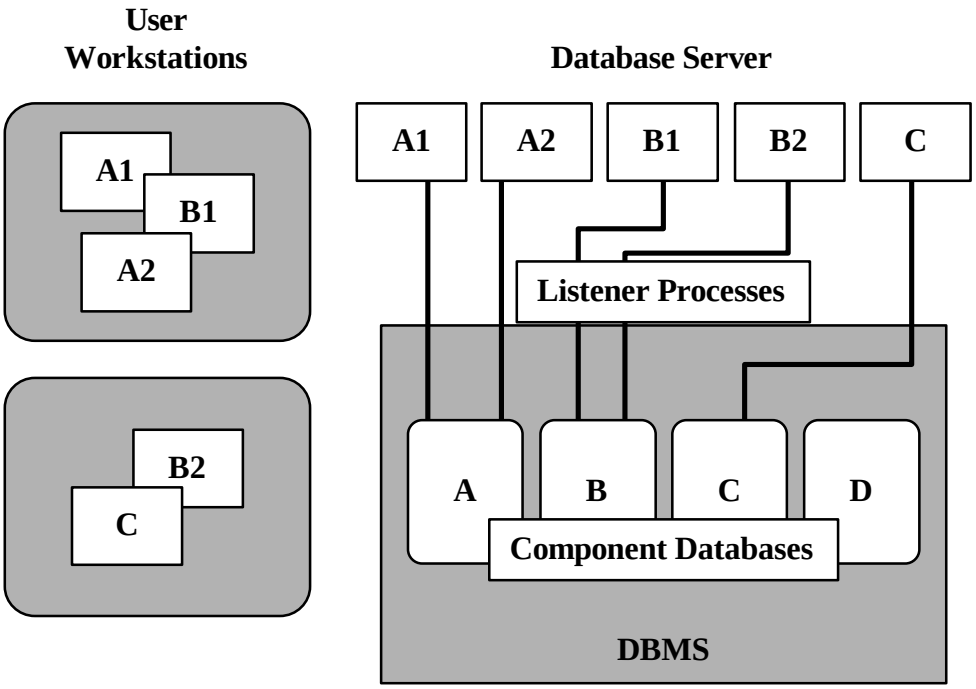

Figure 4-3: Session Management

4.2.5.2 Discretionary Access Control

Discretionary access control is used to manage users' permissions to employ applications to access or modify data managed by the SDS. It has a broader scope than information security. Security is focused on whether users are permitted to know about and allowed to view certain information. Discretionary control of access deals not only with users' permissions to change information but also the context in which they are permitted to make changes. Users will have access to multiple databases through many different applications. Their overall database permissions are the union of the permission sets of the individual applications they have the right to use. At any point in time only the subset of those permissions relevant to the active session can be allowed to be active.

Figure 4-4 illustrates the need for discretionary access. A user has three database sessions active, one with each of three different applications. Each application accesses a different set of objects within the database. The data objects shown represent all objects that a user has permission to access and are marked to show which application context is relevant to that access. If all of the user's database permissions were active at all times, it would be possible for one or another of the applications to access and modify data that is not relevant to it. Instead, each application must only be able to access its corresponding data objects.

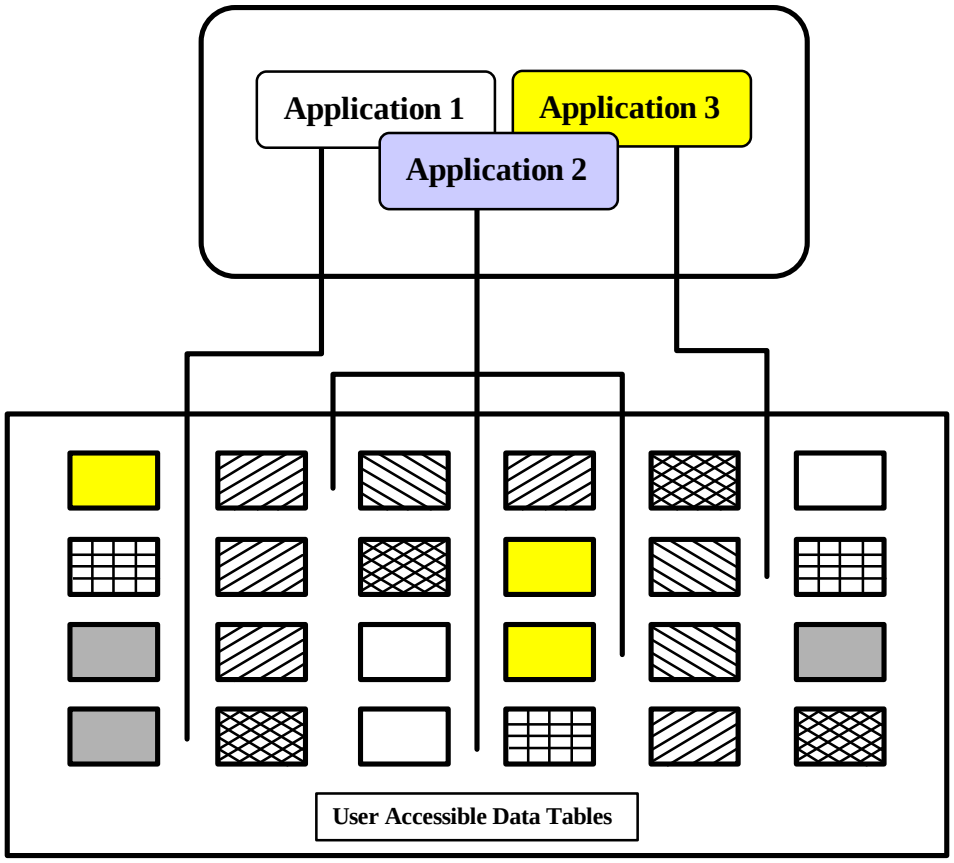

Figure 4-4: Functional Context

It is the responsibility of database and application developers to provide discretionary access controls. The operational sites' administrators are responsible for using those controls when assigning database and application privileges to users. The necessary access controls will be defined in the database segment design as discussed in subsection 4.2.5. Session management by the DBMS provides the database and application developer the isolation needed to implement discretionary access. When designing access controls the following principles apply:

- Users shall have unique accounts within the DBMS. Those accounts shall have only the database permissions needed for their work

- A user's database permissions will only be active within the context of the current application and database session. In other words, when a user starts a database session through some application, that session will only be able to access the data objects appropriate to the application and the only active permissions on those objects will be those appropriate to that application's use of those objects.

These context-specific controls are necessary because users will have access to multiple applications and each application has its own set of database permissions. As a user is granted access to data objects based on the applications needed, the total set of database grants for that

user expands. The DBMS manages sessions at the user account level, so each user has all granted permissions on all objects whether they are relevant to the current session or not. If access were not dependent on context, users could have inappropriate permissions for a particular session. For example, a user might be able to write to a database segment that is supposed only to be read by the current application. Such pathological connections to data objects will, almost inevitably, lead to data corruption.

The context in which an application operates on the database is the application's "database role." A *database role* is the minimal set of database permissions needed for an application to function correctly. Since these roles are linked to the application, their definition is the responsibility of the application developer. However, the roles are implemented within the database so they become part of the database segment. Role implementation is discussed in subsection 4.3.5.

4.2.5.3 Access Management

Access management addresses the work of system and database administrators giving users the permissions they need. They must be able to connect users to applications, to databases, and to database segments. They must also be able to revoke or modify those connections as users transfer or assume different responsibilities. The large number of applications and databases available within COE-based systems could make the administrators' tasks unmanageable if access management is not supported with the proper tools. This section discusses the developers' responsibilities for supporting access management.

The act of adding an application to a user's access list, menu, etc. entails adding associated database permissions to that user's DBMS account. Similarly, revoking access to an application requires that corresponding privileges be revoked within the DBMS. Users must have the proper permissions on both the application and the database, so the two system activities are interdependent. Access to applications will often be granted in logical sets or groups of related applications. As discussed in the previous section, access to databases must be linked to each individual application or the functional context of the application is lost. One application could have multiple permission sets if the same executable is used for both read-only and read/write accesses.

The grant association process is illustrated in Figure 4-5. A user is being given permission to use three applications. As a part of that process, the user must also be assigned the database roles associated with those applications. Through the database roles, the user receives the permissions on the data objects needed to use the applications. If, later, the user no longer needs to use these applications, the administrators can reverse the process. When the application permission is revoked, the database roles are withdrawn from the user. The other reason for managing database roles at the application level can also be seen here. Assume that these applications represent a group that is accessed together and that have identical database permissions. If the grouping of applications changes at some point, the collective role might not be valid. In addition, if there is not a direct one-to-one correspondence between applications and database roles, it becomes impossible to determine when a database role should be revoked.

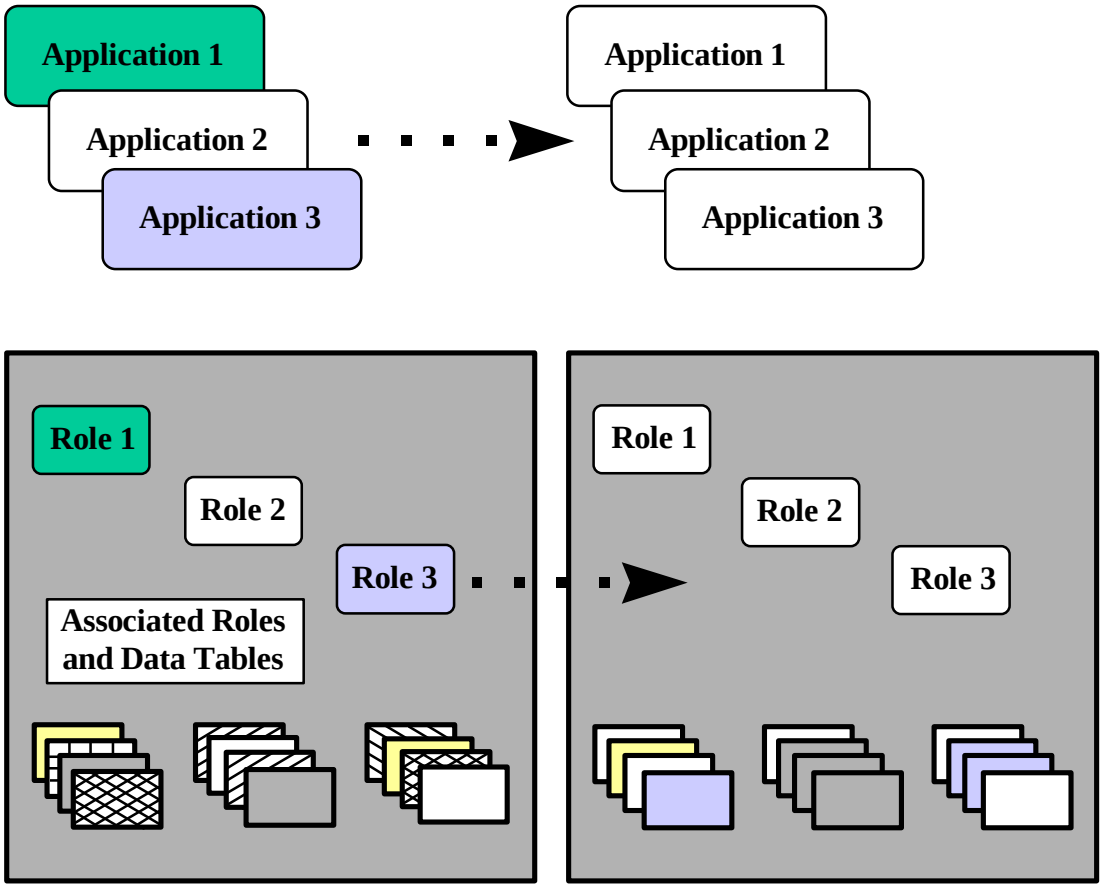

Figure 4-5: Grant Association Process

Database application developers are the only ones with comprehensive knowledge of interactions with the database. They must define the database roles and provide the scripts or command sets that create them for inclusion in the database segment. The scripts that grant and revoke database roles are part of the application segment. This allows them to be executed by the system's administrators when they are managing access to the applications.

4.2.6 Supporting Multi-Database Tools

Access to multiple databases is one of the major benefits that the COE brings to its users. Database browser tools, such as APPLIX, allow users to construct *ad hoc* queries that span different subject areas and that are not supported by mission-specific applications. At the same time, however, such multi-database applications present special problems in the COE context. If the databases were read-only, browsers would not cause problems. However, many databases are designed to be maintained interactively using the applications associated with them. This means that users will have permission to write to databases. Those write permissions are potentially active when a user is using a database browser that means that the browser tool can also write to COE/SHADE databases. This is the reason for the database roles discussed above. Since the

browser is independent of the applications designed for particular databases, it will be unaware of any constraints or business rules that are in those applications. Thus it could corrupt data due to its ignorance of the rules.

The key to ensuring database integrity in this case (as in all others) is the enforcement of constraints and business rules within the database, not within the database applications. If the rules are part of the database, they cannot be bypassed. While the SDS may withhold write permissions from browser tools, maintaining the constraints in the database provides an extra measure of protection. This also supports the future employment of browser tools as multi-database read/write applications.

The second issue is one of understanding the context of a particular database. When users formulate queries that span multiple databases, they are likely to encounter differences in the way information is represented among those databases. This could lead the users to draw erroneous conclusions from their query results because they do not understand the differences between the databases. To limit the chances of this, component database developers shall provide comprehensive information on their databases to be incorporated in the DBMS data dictionary and the SHADE repository. This information is part of the database segment. At a minimum, developers must provide comments for each data element and object (including triggers and stored procedures) that explain its usage and (where appropriate) units-of-measure.

4.2.7 Client/Server Independence

The COE uses a client/server architecture. This applies to database services as well. Developers must preserve the independence of their applications, functioning as DBMS clients, from the SDS. Specifically, applications that access databases must not be built so that they have to reside on the SDS in order to work correctly. It cannot be assumed that all operational sites will have a local SDS. Further, where sites have a local SDS it may be on a separate machine that is dedicated to the DBMS, or the server may be collocated with the application on a single machine acting as the application server and the SDS. To maintain independence and support the client/server architecture, applications cannot assume they reside on the SDS.

To sustain the independence between DBMS clients and the SDS, developers must not mix extensions to the DBMS with their databases and must separate the database from the applications that use it. If specialized data management services are needed by particular applications and are not part of the COE database services, the provision of such services must be approved by and coordinated with the DISA Chief Engineer.

For example, assume some application needs a COTS expert system shell to manage a knowledge base that is a component of the application and that interacts with the SDS. The expert system shell, to work properly, has to be collocated with the DBMS. The expert system then becomes a segment that is separate from the application that uses it.

4.2.8 Distributed Databases

A distributed database is one whose data is spread across multiple sites. Data is replicated in a distributed database when copies of particular objects or records exist in more than one of those sites. Data is fragmented when they are split among sites. Databases are distributed (fragmented or not) to improve responsiveness and increase availability in systems that serve geographically dispersed communities. Databases are replicated to enhance their survivability in the same circumstances. In either case, one implementation objective of any distributed database is to provide location transparency. This means that the user need not know where data is located to be able to access or work on them.

Depending on the component database, the COE/SHADE has several flavors of distributed databases. Some current databases are relatively static and are replicated at multiple sites, but exist independently. They are updated through the periodic replacement of information at each site that has a copy. Others, such as the JOPES Core Database, are dynamic and are replicated concurrently across several sites for survivability. They use transactions to effect updates at the affected sites. Some systems, like the Air Force's Theater Battle Management Control System (TBMCS), both replicate and distribute data on multiple servers within a site to distribute processing and enhance survivability.

The COE provides distributed database management services for the developers of distributed databases so they can maintain location transparency and distributed transaction processing. The specific services implemented for a particular COE database system will depend upon the nature of its distributed data and are the responsibility of that system's Chief Engineer. GCCS, for example, uses an asynchronous transaction model. A financial system may require the use of the more restrictive, but synchronous, two-phase commit.

The technology that supports the distribution of databases as used in the DII COE is evolving rapidly. The GCCS program, for example, does not at present prescribe a particular implementation method. Developers of distributed databases must coordinate their activities with the DII COE Chief Engineer and their program's Chief Engineer to ensure that their approach can be supported and is consistent with the objectives of the broader program. When a distributed database is implemented, developers should keep in mind that the distribution plan (fragmentation schema) may change over time. Distribution methods and the tools used to support them will also evolve as technology matures. Where developers are assigned responsibility for database fragmentation schemas, each fragment shall be in a separate segment so different schemas can be implemented.

The distribution of data also means that users may have access to multiple SDSs. The assignment of users to servers will depend on the distribution schema as implemented for the various sites. The sites' DBAs are responsible for aiming users' processes at the correct SDS. Developers shall not assume that users are attached to a particular server. Developers' applications shall not modify the user's DBMS environment to associate them with a particular SDS. The COE/SHADE data access tools facilitate transparent access to distributed data.

4.2.9 Backup and Recovery

Database backup and recovery address the protection and preservation of information in SHADE databases for DII COE-based systems. The current discussion addresses only those backup and recovery issues specific to databases and database management systems. Conventional system backup and recovery are addressed with the appropriate common support tools for the system as a whole.

COTS DBMS software provides sophisticated tools to prevent data loss or corruption due to system or media failure. Their tools are focused on transaction management with the overriding goal of ensuring that the database can be recovered to a consistent state no matter when or how failure occurs. In the most general sense, DBMS recovery mechanisms maintain transaction logs that keep a continuous record of all database changes. In the event of a system failure, those logs are applied to the database to remove incomplete transactions and recreate committed ones. In the event of media (disk) failure, the last archived copy of any affected DBMS files is used together with the transaction logs (archived and online) to 'roll forward' or recreate all changes that are not in the restored DBMS file.

Execution of DBMS and database backup and recovery is the joint responsibility of a site's system and database administrators. DISA provides tools to assist them in executing their duties. COE developers must implement their databases within the constraints of the DII COE tools and the DBMS vendor tools. Support for special requirements, such as off-line archiving of transaction logs, must be coordinated with the sponsoring COE program office and the DII COE Chief Engineer.

4.3 Guidelines for Creating Database Objects

This section provides guidelines for developers in creating their database segments. Its objective is to support consistency across different databases and improve the mutual independence of the database federation. Also, the guidelines strive to make sure database segments do not inadvertently affect each other.

Developers should strive to make all object names meaningful. Names must start with a letter of the alphabet and may include letters, numbers, and underscores. Names may be 1 to 30 characters in length (except for table names that are restricted to 1 to 26 characters) and cannot be a DBMS reserved word (refer to Appendix F for a list of these reserved words). Case does matter when creating names in the DII COE environment. While a specific RDBMS (such as Oracle) may not be case sensitive in naming objects, there are some (such as Sybase) which are case sensitive. To ensure consistency and portability of database objects and their elements, database object and element names must be implemented in uppercase.

4.3.1 Database Accounts

Three categories of database accounts have been defined within the COE: DBAs, Owners, and Users. They have different functions and levels of access to the DBMS based on those functions.

4.3.1.1 Database Administrators

The Database Administrator (DBA) accounts have access to all parts of the DBMS. They are to be used only for system administration. Their use by database segments is prohibited except during the installation process as discussed in Chapter 5.

4.3.1.2 Database Owners

The Database Owner (DBO) accounts are the creators and owners of the data objects that make up an application's database segment. The name must be unique within the COE community and approved by the SHADE Chief Engineer to avoid naming conflicts. Developers will normally use the segment prefix or a variation of it as the owner account name. The segment prefix will also be used as the database schema name and will be incorporated in database file names as discussed below. Owner accounts must have their password changed after a database installation. Users shall not use the owner accounts to connect to databases. Developers shall not grant the DBA privilege to owner accounts.

All of the database segment's installation, except the definition of physical storage, the creation of the DBO, and the creation of database roles, must execute using the DBO account and password. After a successful installation of the data store segment, the DBO account's password must be changed and its connect capability must be disabled.

4.3.1.3 Users

User accounts belong to the individuals accessing COE databases. Each individual must have a unique user account. User account naming conventions are defined by the individual COE program office (e.g. GCCS Chief Engineer) and will usually be the same as the user's operating system account. The user's account name must be unique within a specific COE database server and may be required to be unique within a COE program. Creation and maintenance of user accounts are a site-DBA responsibility within the rules provided by the specific COE program office. Developers shall not assume the existence of particular users and shall not create user accounts.

The creation of accounts that perform database services is an exception to the rule that developers not create user accounts. Such accounts support autonomous processes, such as message parsers, that access a database on their own. These processes cannot connect to a database using the DBO account for reasons of security and data integrity, but their identity must be known to developers for their specialized database permissions to be set up correctly. Such accounts will be defined by the developer and created as a part of the segment installation.

4.3.2 Physical Storage

Database management systems provide file management transparency across multiple host computer systems by hiding the details of file storage from the database's data objects. At the same time, however, the placement of data objects on physical storage devices has an effect on system and database performance due to disk contention and other file system access issues.

Developers cannot assume that DII SDSs have uniform hardware configurations. Some will have disk arrays, possibly mirrored, that appear as a single, large mount point; others will have multiple mount points representing separate disks or several mirrored arrays. Further, it cannot be assumed that existing hardware configurations will remain static or that current disk-mirroring technologies will remain in use. DII developers, therefore, shall not use 'raw' partitions, but shall place all files in their segment's directory tree. DISA will provide software tools to insulate developers from the SDS's physical implementation. DISA is responsible for providing the core configuration for a COE database server. The site's administrators are responsible for configuring installed servers for optimum performance.

4.3.2.1 Data Store/File Standards

The DBMS-managed components of a database segment can be grouped into functional sets based on their use within the segment. These functional sets are defined as a data store. Data stores are physically kept in database files whose implementation varies depending on the DBMS being used. A segment's database will normally consist of two functional sets (data and indexes) and hence two data stores. The data store identifier will incorporate the database segment prefix and the function of the data store. `GSORTS_DATA` is an example of a data store name.

Developers shall define one or more data stores for their database segments. The objective is to allow data files to be spread across multiple, physical storage devices based on the data store's function within the DBMS. Data store names must be meaningful and use a maximum of 30 characters (uppercase letters, numbers, and underscores). As discussed earlier, the name is case sensitive and only uppercase letters will be used. No DBMS reserved words will be used.

Data store names must also be associated with the segment and function. Most applications will have either two or three data stores: Data, Indexes, and (if needed) Static data. The following naming convention is to be used:

`<segment prefix>_DATA`,
`<segment prefix>_INDEX`, and
`<segment prefix>_STATIC`

for the three storage areas respectively. The Logs within a Sybase database are treated as data stores in a Sybase implementation.

4.3.2.2 Data Storage Implementation

Database segments shall create their data stores through the segment's `PostInstall` descriptor.

Database segments shall use the `COECreateDS` API to implement physical data storage. This API allocates physical storage and creates the data store. For Sybase, this includes the storage area for logs. `COECreateDS` hides the SDS's implementation of physical storage. In this way the database segment is insulated from the physical server implementation, whether it uses raw devices or file system directories, has disk arrays, or uses other storage techniques. Figure 4-6 illustrates data store allocation.

Where `COECreateDS` is not available, developers will provide the scripts to create their data stores and the operating system files associated with them. Data files will be created in the `DBS_files` subdirectory of the database segment using the API provided by the DBMS vendor. One or more data files may be created to support each storage area. The method for file creation varies with the DBMS being used. See Appendix F for DBMS-specific file creation information. The data file names should be chosen so they are clearly associated with the storage area. The recommended naming convention is

`<segment prefix>_<store type><n>.dbf`

where '*store type*' is the storage area's purpose (e.g. index) and '*n*' is a one-up serial number for the file. An example data file name is `gsorts_data1.dbf`.

CREATE_DS DBSORT < DBSORT_LIST

where DBSORT_LIST file contains:

DBSORT_DATA 1,000K
DBSORT_INDEX 1,000K
DBSORT_LOG 300K LOG

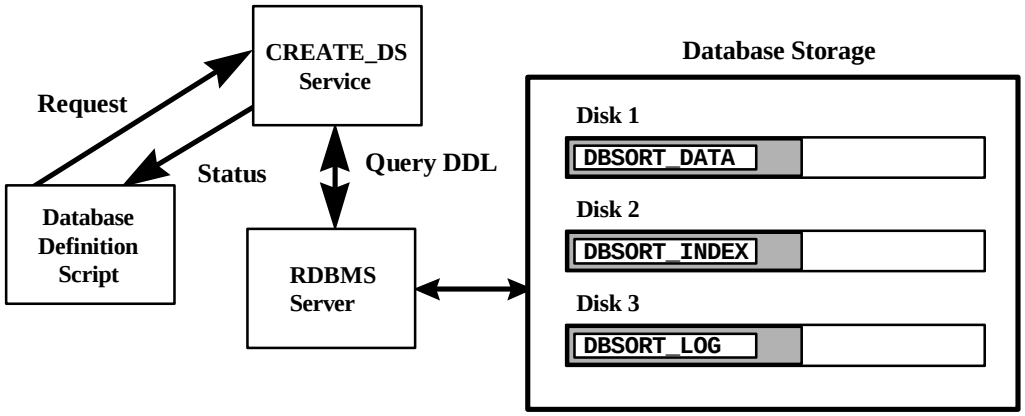

Figure 4-6: Data Allocation

4.3.3 Database Definition Scripts

A *database definition script* is a shell script that contains all database definition commands for a specific database object. These objects include tables, views, triggers, and stored procedures. The name of the script is the same as that of the database object it defines. Depending on the object type, multiple sections can be defined within one file to perform all the data definition functions required for that specific database object.

The scripts used to create data objects are also used by database administrators in the maintenance of the databases and the SDS. DII database administrators have to manage thousands of data objects (tables, views, etc.) spread across multiple database owner accounts. Routine maintenance such as rebuilding corrupted indexes or views can become impossible because the DBA cannot locate the script file that contains the object's definition among the thousands of scripts on the SDS. To avoid these problems, DII developers must organize their data definition language (DDL) commands into a series of database definition scripts. These scripts must conform to a particular file naming convention and structure.

The database definition script is structured to execute various database definition commands based on the input argument given to it. This functionality is implemented with a case statement that executes on the input argument. Table 4-1 lists the input arguments to be used for database definition scripts. See Appendix F for examples of data definition commands.

A database definition script for a table should contain all constraints, triggers, and indexes for the table. Legacy views (see subsection 4.3.4.3 below) of the table, if authorized by the cognizant configuration management authority, may be included in the table's definition script. Other views must be created using their own scripts. Rules, stored procedures, packages, and other objects should each be created in their own separate scripts.

Database roles that are associated with a database schema, such as a default read-only role, can be provided with a database segment's definition scripts. The grants for that role, since it is part of the segment, can be incorporated into the table and view definition scripts. Grants to database services accounts (e.g. message parsers) can also be incorporated in those scripts. Database roles that are associated with applications or those whose grants span multiple database owners must be created using their own scripts. These scripts should include all the grants needed for the role regardless of the object's owner. Such grants should be segregated by owner.

The `CREATE_DATA_STORE` argument for a database definition script should only be used when the COE tool `COECreateDS` is not available.

4.3.4 Database Objects

The definition of a database schema – the set of data objects, their interrelationships, constraints, and rules for access or update – is the responsibility of the developers. Developers of application database segments shall not duplicate data objects that are part of the corporate databases provided by DISA. Where possible and appropriate, developers should take advantage of and share objects belonging to other databases as found within the SHADE repository. If a database segment does not meet the full needs of the developer, changes should be proposed to the cognizant configuration authority for the database segment meeting most of the needs. The developer may choose to develop a similar database segment, pending resolution of the change request. To facilitate sharing of data, developers shall provide the definitions of their schema components for inclusion in the DBMS data dictionary as discussed in section 4.1. Definitions for data stores, tables, elements, stored procedures, and views are stored in the system's data dictionary tables as comments. The maximum length allowed for the description is 255 characters. In addition, narrative information on all these databases should be provided during Segment Registration so developers can access their definitions in the COE online services (see Chapter 10).

Developers shall provide DISA with their proposed database schema early in the segment design process. The schema will be reviewed for duplication of objects in other component databases. See Chapter 3 for more information on the database segment development cycle.

4.3.4.1 Database Tables

Database tables are the objects that store data records. Within a database schema, data elements will be logically grouped to form tables. Table names must be meaningful and a maximum of 26 characters in length (uppercase letters, numbers, and underscore). This size differs from the 30 characters available to other objects because of the legacy view naming convention (see

section 4.3.4.3). If Oracle database snapshots are being used for data replication services for other sites, developers should limit the table name to 20 characters. Oracle will use the remaining six characters to identify the internal tables and views that support a database snapshot. No reserved words may be used in the table name.

Argument	Purpose
CREATE_DATA_STORE	create a data store (use CREATE_USER to create the DBO account first)
DROP_DATA_STORE	remove a data store
CREATE_ROLE	create a database role
DROP_ROLE	drop a database role
CREATE_RULE	create a Sybase rule
DROP_RULE	drop a Sybase rule
CREATE_TABLE	create a database table
DROP_TABLE	drop a database table
CREATE_VIEW	create a database view
DROP_VIEW	drop a database view
CREATE_CONSTRAINT	create a database constraint (i.e. foreign key)
DROP_CONSTRAINT	drop a database constraint
CREATE_INDEX	create an index
DROP_INDEX	drop an index
UPDATE_INDEX	perform update statistics for Sybase indexes
CREATE_USER	create a database user account
DROP_USER	drop a database user account
DISABLE_LOGIN	Revoke connection privileges (login) from a database account
ASSIGN_GRANTS	assign grants to a user or role/group
REVOKE_GRANTS	revoke grants from a user or role/group
LOAD_DATA	load a table with data (from within the command script)
CREATE_PROCEDURE	create a stored procedure or database package
DROP_PROCEDURE	drop a stored procedure or database package
CREATE_TRIGGER	create a database trigger
DROP_TRIGGER	drop a database trigger
CREATE_SEQUENCE	create an Oracle sequence
DROP_SEQUENCE	drop an Oracle sequence
REGISTER_DATA	load application data (i.e., configuration parameters)

Table 4-1: Definition Script Arguments

The creation of tables in System-owned storage areas (e.g. the Oracle SYSTEM tablespace or the Sybase master database) is prohibited. The tables must be created in storage areas created by and belonging to the segment. When creating a table, the storage area name must be specified. “Create

table" statements must stipulate `NOT NULL` or `NULL` constraints for each column because different DBMSs may default differently on this constraint type.

4.3.4.2 Data Elements

Data elements are the columns or fields within a schema that are grouped together into tables. Data element names shall comply with DOD standards from the DOD Data Model (DDM) and Defense Data Dictionary System (DDDS) where applicable. Within a schema developers should use the same characteristics (data type, length, number precision, default values, constraints, and definition) for all occurrences of the same element name. If elements are chosen from the DDM, they shall use the data type and units of measure prescribed in the standard.

Developers shall not use data types that are machine-dependent. This applies primarily to numeric data. Data elements may be shared across tables and data stores, and across COTS DBMS servers. As an example, the 'float', 'double', and 'real' data types are machine-dependent in both Oracle and Sybase. See Appendix F for more information on data types available in specific DII COTS DBMS and which are machine independent.

The use of default values and declarative constraints is recommended to ensure data integrity and consistency. Developers must balance this against instances where invalid data items must be forced into the system, especially when dealing with real-time data. In such cases declarative constraints could cause data loss.

4.3.4.3 Database Views

A view is a stored query that does not actually contain or store data, but derives its data from the tables on which it is based. These base tables can in turn be tables or views. View names must be meaningful and a maximum of 30 characters in length (uppercase letters, numbers, and underscore).

Views are often used to restrict users' access to vertical (columnar) or horizontal (row-wise) subsets of data tables. Views can also be used to hide data complexity when displaying related information from multiple tables or to present data from a different perspective than that of the base table. Views can provide location transparency for local and remote tables in a distributed database, a convenient way of storing complex queries, and isolation of applications from changes in definitions of base tables.

Views can be queried, updated, inserted into, and deleted from, with restrictions. All operations performed on a view affect the base tables of the view. Current DBMSs are limited in their ability to support updates through views. If developers need updateable views, the DBMS's capabilities and restrictions must be kept in mind. If the updateable views are required for security or data privacy, developers should not grant users access to the base tables, only to the views.

In general, the following restrictions apply to updateable views.

- **Horizontal (row-wise) views:** SDSs can support inserts, updates, and deletes through horizontal views. Such views include those where one table is used to constrain the view to a subset of rows in another table. Developers are responsible for implementing appropriate error handling if users try to insert a row that duplicates a hidden row or that contains a value in the restricting column(s) the users are not permitted to see.

- **Vertical (columnar) views:** SDSs can support updates and deletes through vertical views as long as the database constraints do not reference hidden columns. Inserts can only be supported if all hidden columns are allowed to be null or if triggers are provided to populate them with default values. Developers are responsible for implementing appropriate error handling if a user's update violates a constraint on a hidden column.

- **Multi-table views:** At present, the SDSs implemented in the COE cannot consistently support data modifications through views of more than one table. Developers should implement such updateable views in applications. These views should be accompanied by comparable read-only views of the individual tables.

A view is dependent on the objects referenced in its defining query. All of these objects must exist, and the required privileges to these objects must have been granted to the owner of the view before the view is created. Views will be created in a database segment as part of its install process.

Legacy Views are views created to support applications written against earlier versions of a database object. Such views make the object appear as it did in an earlier version of the database segment. They support read-only legacy applications. Applications that need to update data will not be able to use legacy views. Code modifications required to update data in the new data structures will need to be coordinated with data structure changes.

The decision whether to require Legacy Views rests with the DII COE Chief Engineer working with the affected program's Chief Engineer on a case-by-case basis, although developers may choose to do so on their own. In most cases, DISA will require Legacy Views only for Shared or Universal public databases whose tables support a large number of read-only users.

When Legacy Views are provided, they must be implemented in the following manner. When a database table is created, a view that maps directly to the table must also be created. When the database table is modified, the view of its previous version is also modified so that applications accessing the view are unaware of them, and a view that maps directly to the new structure is created. This method allows applications that access the table to continue to operate if immediate source code modifications are not possible. Applications must eventually be modified, but in the meantime views can be maintained to support previous versions of the table. Figure 4-7 demonstrates the use of legacy views across four versions of a database table.

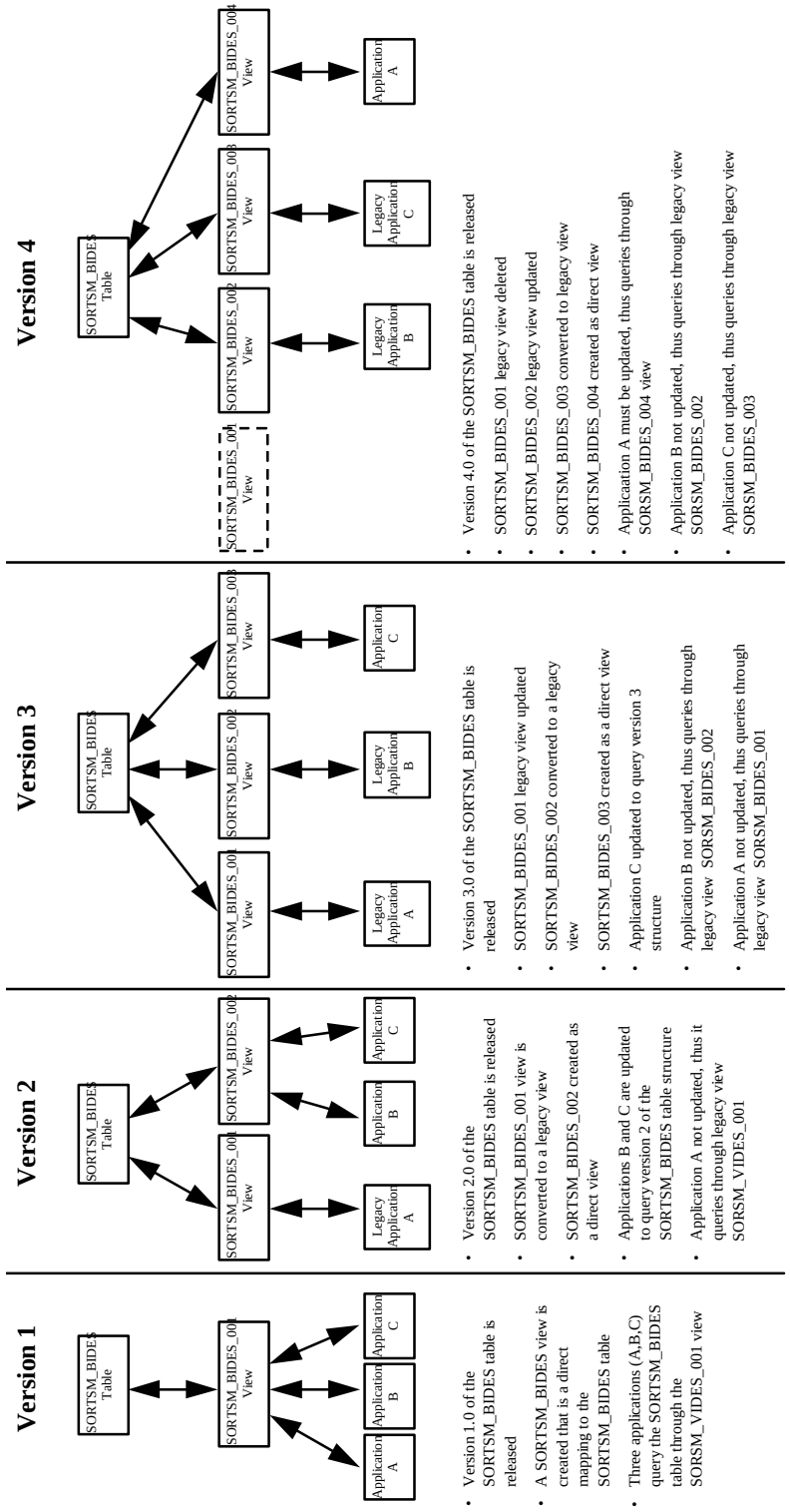

Figure 4-7: Legacy Views

View names for legacy views shall consist of the table name followed by a three-character table sequence number. They will be a maximum of 30 characters (uppercase letters, numbers, and underscore). Example legacy views are `NID_ACFT_003` and `IDBIND_001`, where the first is a view representing the third release of the NID aircraft table and the second is a view representing the first release of the IDB individuals table.

4.3.4.4 Rules on Database Objects

Rules on database objects incorporate several different concepts. Their underlying purpose is to maintain database integrity through the enforcement of the constraints and business rules of the database.

For purposes of this document, the following definitions apply. *Constraints* are restrictions on data elements with respect to the values they may contain. For example, a country-code data element could be constrained to the set of DIA-prescribed two-character country codes. *Business Rules* are restrictions that occur in the context of database operations that affect multiple interrelated objects and elements or that are beyond the ability of a constraint to express them. For example, any update to a facilities table may require that an entry be written to an audit table recording the ID of the user making the change and the time at which it was made.

Within the SDS, developers may use DBMS constraints, stored procedures, rules, packages, or triggers to implement either constraints or business rules. The choice among these will depend on the capabilities of the COTS DBMS being used.

4.3.4.4.1 Constraints

Developers should define all entity integrity constraints and referential integrity constraints that apply to their database schemas. The information in these constraints is vital for maintaining database integrity. Entity integrity should be enforced whenever possible using default values (not null values), unique values, and check constraints. Domain Keys (e.g. the SQL Check constraint) should be used to maintain the validity of column values. Unique columns should be constrained rather than indexed. While the DBMS may use an implicit index, as Oracle does, to enforce uniqueness, defining the constraint clearly documents the database design for the users. Database primary keys, foreign key constraints, delete cascade actions, and update/delete restrictions should be used to maintain referential integrity.

Primary and foreign keys convert logical relationships that are implicit in the database design into explicit relationships. Primary keys identify unique physical records. Foreign keys relate primary keys to data in other tables by requiring each value in a column or set of columns to match those in a primary key in the referenced table. Foreign key constraints enforce referential integrity by preventing invalid data entry into the database tables.

Where appropriate, constraints should be used to supply default values for columns. The NULL/NOT NULL constraint must be explicitly stated for each column in all tables because different DBMS implementations may behave differently with respect to nulls.

Constraints must be explicitly named. Constraint names must be meaningful and must not use reserved words or default names. They may not exceed 30 characters (uppercase letters, numbers, and underscore). The recommended naming convention is

`<table name>_<cons>`

where '*table name*' is the name or abbreviated name of the table or table and columns involved in the constraint and '*cons*' is PK for a Primary Key, FK for a Foreign Key, or CK for a Check constraint. Foreign key constraint names should incorporate references to both tables. Examples of constraint names are `IDBF_PK`, `EWIRD1_EMIT_FK`, and `ACFT_USR_CTRY_CK`.

In most cases developers will wish to create their constraints after the data fill has been completed in order to speed up the fill process. The implicit index that accompanies a Primary Key or Unique constraint will slow the data fill significantly. Constraints should not normally reference data objects that are outside the database segment. See below for methods to implement inter-database constraints when they are needed.

Constraints should still be included in a database segment even when they cannot be enforced. If developers must allow invalid data items into their database, as may be the case when processing real-time data, they may not be able to enforce declarative constraints without losing information. Constraints should still be defined, but disabled (e.g., by using Oracle's disable constraint command) so that users and administrators can understand the database schema. The `ReleaseNotes` segment descriptor and the comments on the object stored in the data dictionary shall state that these constraints are deliberately disabled so the site's DBAs know that it is intentional. If constraints must be left disabled, the developers are responsible for providing tools that support cleanup of invalid items.

4.3.4.4.2 Stored Procedures

Database stored procedures and functions consist of a set of DBMS commands (e.g. SQL statements, and Oracle PL/SQL or Sybase Transact-SQL constructs) that are stored in the database and can be invoked by an application to perform a task or a set of related tasks. Stored procedures and functions can be used to obtain tighter control of database access. In addition, they improve performance by reducing the amount of information that travels over a network and because they do not require interpretation prior to their execution. The use of stored procedures and functions also reduces memory requirements as only a single copy is loaded into memory for execution by multiple users.

Stored procedures are used to maintain database integrity or to enforce business rules when the constraints imposed are too complex for simple SQL constraints. These procedures are stored in the database and can be executed from any environment in which an SQL statement can be issued. A maximum of 30 characters (uppercase letters, numbers, and underscore) may be used for the stored procedure name. Procedure names should incorporate the name of the object(s) they modify and some meaningful indication of their functions without using reserved words. Two examples of stored procedure names are `NID_UPDATE_PROC` and `GSORTS_FETCH_UNITID`.

Stored procedures should not normally reference data objects that are outside the database segment. See below for methods to implement inter-database stored procedures when they are needed.

Database stored procedures are installed after all of the database objects defined in the database segment have been installed. In general, the stored procedures in a database segment should support integrity checks that are typically invoked by triggers. A database segment may also provide stored procedures that perform standard access functions against the segment's tables. These access functions can provide better performance and reduce maintenance efforts if underlying structures are changed.

4.3.4.4.3 Triggers

A database trigger is a procedure that is automatically executed when a triggering event occurs on the associated table. A trigger can only be defined on a table and will fire whenever the associated event occurs on the table or a view of that table. The action of a database trigger may cause another database trigger to fire. Triggers can be used to generate derived column values, implement complex security rules, perform auditing, maintain table replication, prevent invalid transactions, and enforce referential integrity.

Most triggers will be used to maintain database integrity. Others may be used to signal or send data to other, interrelated or dependent database segments. Triggers may also be used to support the proper replication of data and to perform data conversions. They are not to be used to start application processing based on data entry. Trigger names must be meaningful (the table name and trigger type should be part of the trigger name) without using reserved words. They may use a maximum of 30 characters (uppercase letters, numbers, and underscore). An example of a trigger name is `TAC_REMARKS_UPD_TRIGGER`.

Triggers should not normally reference data objects that are outside the database segment. Database segments should not install triggers on data objects outside the segment. See below for methods to implement inter-database triggers when they are needed.

Database triggers are installed after all of the database procedures are installed. This order is prescribed because triggers may invoke stored procedures. A trigger's body may contain DBMS commands (e.g. Transact-SQL or PL/SQL blocks) or it could invoke stored procedures to perform the same functions. The use of stored procedures to support triggers is recommended for performance and maintainability.

4.3.4.5 Indexes

An Index is an optional structure associated with a table that is used to quickly locate rows of that table or to ensure that a table does not contain duplicate values in specific columns when a uniqueness constraint cannot be used. Indexes speed up retrieval when applications query a table for a range of rows or for a specific row by providing a faster access path to data. Indexes are logically and physically independent of data. The creation or deletion of an index may occur at any

time and does not affect the data stored in the associated table. Furthermore, creation or deletion of indexes only affects the speed of data retrieval, but does not prevent any applications from functioning. Once created, indexes are maintained by the RDBMS and are automatically updated when the data change due to addition, deletion or modification of rows. The presence of many indexes on a table decreases performance when inserting, updating, or deleting data as the associated indexes must also be updated. Indexes also require storage in the DBMS; the use of multiple indexes requires more storage.

Index names must be meaningful without using reserved words. A maximum of 30 characters (uppercase letters, numbers, and underscore) may be used for the index name. It is recommended that the index name incorporate a reference to the table and column for clarity. Developers should review the capabilities of the DBMS before indexing small tables (less than 4000 rows). Indexing small tables can actually hurt performance if the DBMS searches the index instead of reading the entire table into memory. The DBMS's query optimizer may ignore indexes on small tables. Indexes should not be used in place of Primary Keys or Uniqueness constraints.

When considering a column or group of columns for an index, keep the following guidelines in mind:

- Indexes should not be used in place of primary keys or uniqueness constraints. If the DBMS treats nulls in a manner that prohibits the enforcement of these constraints, developers should use a unique index to maintain data integrity.

- To minimize lock/device contention when insertions occur frequently, clustering should always be performed on a key that is statistically more "random" than other keys and is usable in many queries. This is generally not the primary key. Prime candidates for clustering keys include columns accessed by range or used in Order By, Group By or Joins. For example, Date/Time could be a good index key for event data. Long strings generally make poor indexes.

- Too many indexes can hurt performance of inserts, deletes, and updates.

- Prime candidates for non-clustered indexes are columns used in queries when the data being accessed is less than 20% of the data in the column.

- Keep the size of the key as small as possible to improve index storage and data retrieval.

- Indexes help select statements and hurt inserts/deletes. Consider when most of your operations will use the index and, if so, whether the overhead required for the index is worth it.

- Storage of indexes in a separate data store can improve database performance.

- The ordering of columns in SQL 'where' clauses may affect the behavior of the DBMS query optimizer. Check the DBMS vendor's documentation to identify such effects and use the vendor's evaluation tools to assist in optimizing DBMS commands.

- Consider using the various index types offered by the DBMS. B-Tree indexes are good for range selections and ordered retrieval, but can suffer performance problems when used on large sets of ordered, sequentially appended (e.g. time series) data. Hashed indexes are fast, but do not easily support ordered retrievals. Bit-mapped indexes are efficient for binary fields like sex.

4.3.5 Database Roles

A database role, in the general sense, is a group of access privileges for database objects. These roles implement the discretionary access controls discussed in subsection 4.2.5. Database roles also simplify the management of user privileges within the DBMS. They are created by the database segment developers or the developers of applications accessing databases to define sets of access privileges that can be given to users by their sites' DBA. Role names must be meaningful (the database or application name should be part of the group or role name) without using reserved words. They may be a maximum of 30 characters (uppercase letters, numbers, and underscore). Developers should strive to associate roles and their privileges with the applications accessing the database. Each role should have only the privileges needed by the application it supports.

As discussed in subsection 4.2.5.2, active database permissions should be limited to the minimum set needed for the session in progress. Such permission sets are specific to an application's connection to the database. This means that each application requiring access to any database object must have a well-defined database role that includes only the privileges needed by the application and that the role/group be granted only to users who are authorized to run the application. In this case, it is the responsibility of the database segment which supports an application's software segment to create the specific database role for the application and to connect to the DBA account (see the `COEPromptPsswd` API in Appendix C) to assign the grants on the required objects to the newly created role. The DBA account has all necessary privileges to assign grants on any object to any role.

A Shared database segment must provide generic "read-only" roles because of the dependencies of other segments upon it; it may provide "read/write" roles. Developers may create more generic roles or groups that consist of a set of privileges (such as read-only or read/write) on a group of objects (such as all of the objects in a database or some subset of them). Such roles are usually created to provide read-only access to an entire database for users of browsers or query builder tools. In this case, it is the responsibility of the database segment that creates those objects to also create the generic role/group and assign the grants on the required objects to the newly created role/group. Such generic roles are useful when widely used, large, read-only databases such as the NID must be implemented. Such generic database roles should be used with caution as they may grant applications more privileges than they really need. Generic database roles should seldom, if

ever, be used to grant write permissions. Database developers who implement generic roles or groups must balance the advantages this type of role against the risks of unnecessary or excessive privileges.

Consider the following example. A database segment named TEST has five tables: MASTER, DATA1, DATA2, REF1, and REF2. Two applications, APP1 and APP2, are associated with the segment. The segment should have two read/write roles, TEST_APP1_RW and TEST_APP2_RW, one to support each application. It could also, optionally, have a read-only role, TEST_RO, for users of browser tools. If only one read/write role were created, then users of APP1 could inadvertently modify data that should only be changed using APP2 and vice versa.

When applications are not developed by the database segment developer, the application developers are responsible for creating the roles required to access the database through their applications. The access requirements for such roles must be defined by the application developers and included in the information provided during Segment Registration as discussed in Chapter 3. The permissions required by application's database roles are subject to review by DISA and by the associated database segment's sponsor. These roles will be granted the privileges required to run the application. These privileges may include: delete, insert, select, and update for tables and views; and execute for procedures, functions, and packages. Grants of privileges to roles are discussed in the next section.

In order for the privileges on objects to be assigned to a role, the grantor must have permission to do so, and those database objects must exist. When application developers define database roles to support their applications and those roles are not part of the principal database segment, the roles and the grants that enable them become part of a database segment that is dependent on the database segment or segments that create the referenced objects. See subsection 4.3.7 for more information on inter-database dependencies.

Database roles shall not be granted to DBAs. Their administrative privileges already allow them to grant roles to users without owning the roles. The database roles that are part of the COTS DBMS shall not be altered by developers.

4.3.6 Grants

Grants are the permissions on database objects that allow users to access data they do not own. When a database object is first created, the only account that can access its contents is the owner of that object. Users must be explicitly granted permission to access an object. Privileges that can be granted include: delete, insert, select, and update for tables and views; and execute for procedures, functions, and packages. Privileges that should not be granted include index and alter for tables. Grants allow the DBA to administer and the DBMS to enforce the discretionary access controls required. As discussed in the section on database roles, developers should grant only the minimum set of permissions needed for the applications that access their databases. Grants should be made to roles/groups and not to individual users.

Consider the previous example. APP1 is used to create and modify records in DATA1. It uses MASTER and REF1 as lookup tables. APP2 has the same function for DATA2 using MASTER and REF2 as lookup tables. The read/write role associated with APP1, TEST_APP1_RW, should be granted the select privilege on MASTER and REF1, and select, insert, update and delete on DATA1. It should have no privileges on REF2 or DATA2. TEST_APP2_RW, the read/write role for APP2, would have select on MASTER and REF2; select, insert, update and delete on DATA2; and no privileges on DATA1 or REF2. TEST_RO, for users of browser tools, would have the select privilege, only, on all five tables.

Granting data access to DBMS 'PUBLIC' users is prohibited. Granting data-access privileges to user accounts with the 'GRANT OPTION' or granting administration privileges on database roles is prohibited. Developers shall not make grants of application-level permissions to DBA accounts or to database roles used for DBMS administration. Where segments' applications or databases need special permissions on DBMS objects (e.g. query Oracle's 'v$' tables), the developer must request them from the DII COE Chief Engineer. Such grants should be kept in a separate database definition script (to be executed by the DBA) within the database segment that needs them.

4.3.7 Inter-Database Dependencies

Inter-database dependencies occur whenever database objects in a segment are dependent upon objects in some other database segment. A database object is a dependent object if it references any other object(s) as part of its definition. When a dependent object is created, all of its references to other objects must be resolved. If it has dependencies on non-existent objects, the dependent object may not already have been created or it may have to be validated when the objects it references come into existence. The creation of a dependent object may also fail if its owner does not have the appropriate access to all referenced objects. If the definition of any of the referenced objects is altered, the dependent object may not function properly or may become invalid.

Dependent objects that reference objects created, managed, and maintained by the same database segment do not introduce inter-segment dependencies. In contrast, dependent objects that reference objects in other segments do add complexity to the installation, de-installation, administration, and maintenance of a database segment. Before using dependent objects, developers must balance the advantages of dependent objects against the disadvantages of introducing segment dependencies.

Database segments with intersegment dependencies sometimes benefit from smaller storage and reduced data redundancy. Using data objects that belong to other segments frees up the storage that would otherwise be used for replicas of those objects. When replicated objects are eliminated, changes to those objects need not be propagated across multiple database segments. At the same time, having only one copy of a widely referenced table is likely to increase data quality and currency. Eliminating copies of data objects also reduces the processing load on the SDS by eliminating duplicate updates when changes are made.

Such dependencies also affect the modularity and scaleability of the SDS. Dependent segments must be installed after the database segment they reference. Further, as is the case with other segment types, dependencies can easily propagate when placed on segments that are, in turn, dependent on other segments. Furthermore, when inter-segment dependencies are defined, circular dependencies can be created. A circular dependency exists when two segments depend on each other. In such cases, neither segment can be installed because both require the other to be installed first. If a circular dependency cannot be resolved, then the two segments may have to be merged into a single, larger segment or the dependent code can be moved to a third segment. The dependency of one database segment on another segment's data objects could require the installation of multi-gigabyte databases so that one or two of their tables can be used by some other segment.

Because of the tradeoffs involved in the employment of dependent objects, their use in DII systems is subject to review and approval of the DII COE Chief Engineer.

Where inter-database dependencies are needed they shall be implemented such that the object(s) creating the dependency are owned by the database segments that they belong to. This means that a foreign key constraint belongs in the segment with the table it constrains, not in the segment with the table it references. A post-update trigger added to a table belongs in the segment with that table, not in the segment of the table it updates. Such dependencies may have to be placed in separate database segments that modify the segment owning the object that creates the dependency. See Chapter 5 for more information on segmenting databases that have dependencies. The `Requires` descriptor for such database segments must identify all dependencies on other database segments. In addition, the `Database` descriptor must be used to identify the data object(s) being referenced in other segments so that DISA can choose the most effective segmentation strategy for databases that are widely used.

The following sections describe how developers should implement inter-segment dependencies that may occur through the use of dependent objects, constraints, and database roles.

4.3.7.1 Data Objects

Database segments will have dependent data objects (tables or views) when their information needs can be partially satisfied from tables or views contained in other database segments. If an external table fully satisfies the information needs, it should be referenced directly. Developers may use a dependent view to extract subsets of information from external tables or views or to change the presentation of information (e.g. change units of measure or combine columns). Developers may use views to combine internal tables with external objects to provide information supersets. Also, a table could reference an external object either as a source of constraints or, through a trigger, as a provider of data.

Names of objects created in other schemas must identify the inter-database linkage. Otherwise they are subject to the naming restrictions of their object type. Developers are responsible for ensuring that their object's names do not conflict with those already in the schema.

A table will be dependent on another database segment if its constraints reference objects in that other segment or if it is populated or maintained using a trigger based on an external object. Developers may also create a table that is a superset of an external object to avoid creating and maintaining partially redundant objects. That table would then be combined with a view that joins it with the external object. Developers must use an 'outer join' when defining such a view/table combination unless appropriate triggers are created to prevent decoupling when updates occur to either the internal or external table.

A view that references a table (or view) outside its own segment is dependent on the database segment containing the base table (or view). Once such a view has been created, it will become invalid and have to be recreated if its base table (or view) is modified, renamed, or dropped. Any privileges or synonyms on the invalid view also become invalid until it is recreated.

Developers shall not create indexes on objects in other database segments. Indexes have significant impact on system performance. While they speed retrieval of records, indexes slow updates to tables. The effect of uncontrolled index proliferation could dramatically damage the overall functioning of a DII system. If developers desire indexes on tables in other database segments, they must request them from the SHADE Chief Engineer. DISA will work with the other segments' sponsors and developers to assess the effect of additional indexes. If, based on overall requirements, the request is approved, the segment responsible for the creation of the table will be modified to also create the index(es) required by other segments.

4.3.7.2 Rules in Other Databases

Database segments have dependent constraints or business rules when their integrity constraints or operations involve objects from other segments. Such rules may include foreign keys that reference another schema's tables or triggers that propagate updates based on another schema's transactions.

Any rules – whether they are constraints, triggers, or procedures – shall be created in the schema of the object they are attached to. Names of rules created on other schemas must identify the inter-database linkage as well as the rule's function. Otherwise, they are subject to the naming restrictions of their object type. Developers are responsible for ensuring that their rule names do not conflict with those already in some other schema.

Developers may create constraints in their own schema that reference objects in other database segments. They may not create or modify constraints on objects in other schemas. Such constraints could invalidate otherwise legal updates to the other database. When additional constraints are needed on objects in other database segments, developers must request them from the SHADE Chief Engineer. DISA will work with the other segments' sponsors and developers to assess the effect of these constraints. If, based on overall requirements, the request is approved, the segment responsible for the creation of the table will be modified to also create the constraint(s) required by other segments.

Developers may create triggers and stored procedures or functions on objects in other schemas as long as they do not modify or update the other database's information and do not change the constraints or business rules of the other database. It is permissible, for example, to use a 'post-insert' trigger to copy data from an external data object to one in the developer's database segment. It is prohibited, by contrast, to use such a trigger to change data in that other segment's table.

Excessive use of triggers can result in complex interdependencies that may be difficult to maintain. When implementing a specific function via triggers, developers must keep in mind that a database transaction will rollback if execution of the associated trigger(s) is not successful. Trigger developers must implement exceptions to handle errors or unexpected results that may occur during the execution of a trigger. These exception handlers must ensure that a trigger fails 'open' and allows the owning segment's database transactions to complete regardless of the processing of the dependent trigger. If additional triggers and stored procedures or functions are needed in other database segments, developers must request them from the SHADE Chief Engineer. DISA will work with the other segments' sponsors and developers to assess their effect. If, based on overall requirements, the request is approved, the segment owning the object affected by these triggers, stored procedures, or functions will be modified to incorporate them.

4.3.7.3 Database Roles Spanning Multiple Databases

Developers may need to create roles whose permissions span multiple databases in order to take advantage of their information and to correctly represent applications' information needs. Since database roles implicitly are created at the database server level, which segment they belong to is irrelevant. However, all objects they reference must exist before the role may receive its grants. Accordingly, such roles shall be part of a dependent database segment as discussed in Chapter 5. That segment is dependent on every segment whose objects it references. It must list all of the segments under its `Requires` descriptor. See Chapter 5 for more discussion of the `Requires` descriptor.